Computing RMSD using GPGPUs

By Nicolas Barrios

NVIDIA CUDA Primer

Parallel Computing platform for NVIDIA GPUs

A Bit of History:

- Games and other graphics heavy applications often relied on the graphics processor to do the heavy lifting.
- Researchers/scientists said, “Why not use the graphics processor for **our** computationally heavy workloads? After all, they’re quite similar.”
- Shortly after, NVIDIA officially release the CUDA platform with the express purpose of boosting the performance of these workloads

CUDA Programming Model

While CPUs are developed with general computing in mind, GPUs are purpose-built (down at the silicon level) to run massively parallel applications.

Many, *many* more compute cores than a traditional CPU.

CUDA is an **extension** of C/C++ (there's also a Fortran extension).

CUDA defines keywords and syntax to easier call GPU functions (called "kernels") with multiple threads.

A Simple Example of CUDA

```
// Example from
https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#kernels
__global__ void VecAdd(float* A, float* B, float* C){
   int i = threadIdx.x;
   C[i] = A[i] + B[i];
}
int main(){
   ...
   // Kernel invocation with N threads
   VecAdd<<<1, N>>>(A, B, C);
   ...
}
```

What’s RMSD?

Root-mean-square deviation, also known as root-mean-square error (RMSE)

In the bioinformatics space, it is used to measure the average distance between sets of points defining proteins or other biological structures.

Where d_i is the distance between point i and its corresponding point in the other set:

$$RMSD = sqrt\left(\frac{1}{N}\sum_{i=1}^{N}\delta_i^2\right)$$

Okay so, RMSD + CUDA?

Yes! Thanks to Wolfgang Kabsch.

Kabsch Algorithm: Computes an **optimal rotation matrix** that **minimizes** the **RMSD** between sets of points.

Looking at the sets of points as two matrices, we can take advantage of the matrix multiplication processing that GPUs are built for.

Kabsch Rundown

1. Translate both sets of points in such a way that their centroids end up at the origin of the imposed coordinate system
   a. i.e. with 3D points, we want each centroid to end up at (0, 0, 0)
2. Compute the covariance matrix: $Cov = A^T * B$
   a. Requires a matrix transpose and a matrix multiplication!
3. Compute the optimal rotation matrix
   a. $R = sqrt(Cov^T * Cov) * Cov^{-1}$
   b. Requires 2 multiplications, a transpose, and inverse computations!!
4. Rotate both sets of points using R
5. Compute RMSD

Project Implementation

- Use CUDA's `float3` vector type to simplify initialization and conceptualization of the N x 3 matrices that Kabsch necessitates
- Shared Memory: In reduction and matrix multiplication kernels, allocate enough shared memory for the threads to only access global memory *once*
  - I.e. in matrix multiplication, allocate enough memory to store an entire column of *B* in the calculation of *A* x *B*.
- Each kernel (reducing, transposing, translating, matrix multiplication) is wrapped to be used within C++ files, where the wrapper calculates the necessary # of blocks and # of threads based on the size of the input.

Benchmarking Configuration

- System Specs
  - Intel i7-7700k
  - NVIDIA GTX 1060M 6GB
  - 16GB of Memory
- Compilation Options:
  - `NVFLAGS := -Werror=all-warnings -gencode arch=compute_61,code=sm_61 -rdc=true`
  - `CXXFLAGS := -Wall -Wextra -Werror -std=c++11`
- Generic make rules:
  - ```
    bin/test_%: src/test_%.*
    	$(NVCC) $^ -I$(INC_DIR) -o $@ $(NVFLAGS) -Xcompiler $(subst $(SPACE),$(COMMA),$(CXXFLAGS))
    ```
  - ```
    test_%: bin/test_%
    	./$^
    ```

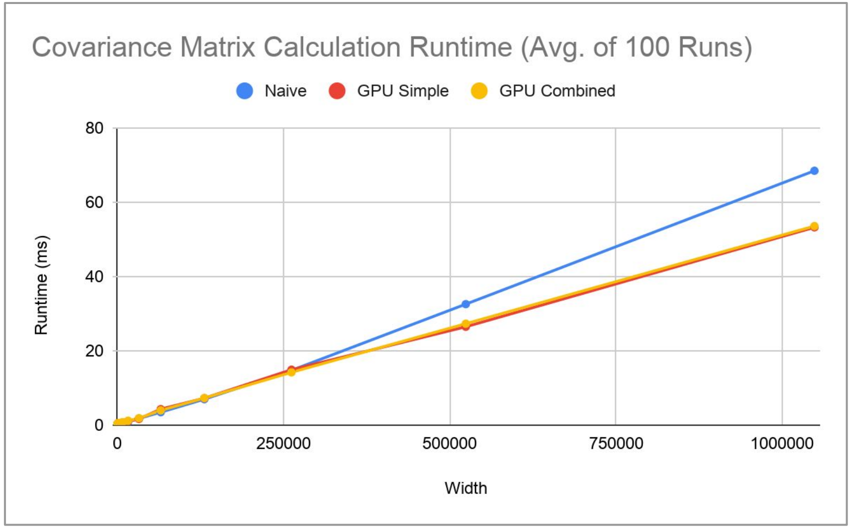
Results

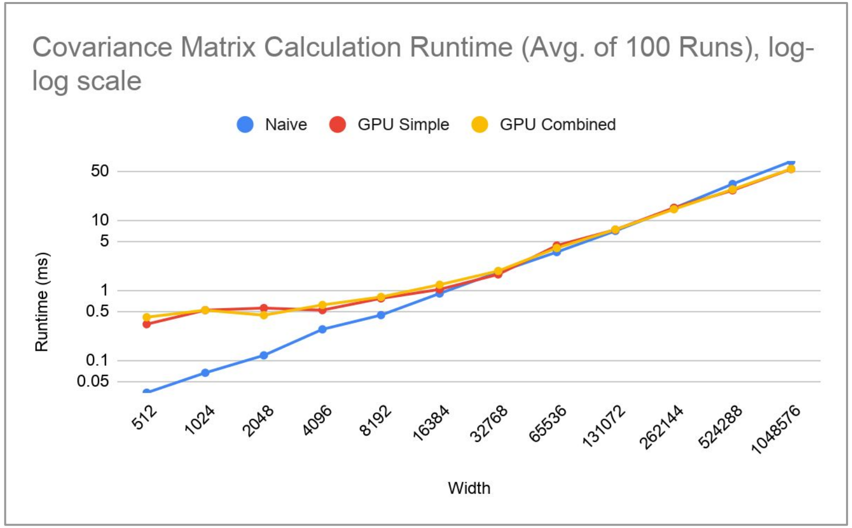
Matrix Calculation Runtimes (ms)			
Width	Naive	GPU Simple	GPU Combined
512	0.0342128	0.325414	0.409593
1024	0.0658724	0.515943	0.516168
2048	0.116834	0.552264	0.437346
4096	0.274821	0.516756	0.612387
8192	0.438935	0.760053	0.796006
16384	0.89224	1.01975	1.19057
32768	1.79857	1.6719	1.87057
65536	3.47928	4.31061	3.97394
131072	6.9812	7.24666	7.29523
262144	14.7412	14.9196	14.2032
524288	32.5542	26.4642	27.2958
1048576	68.4355	53.2135	53.56

Results (cont.)

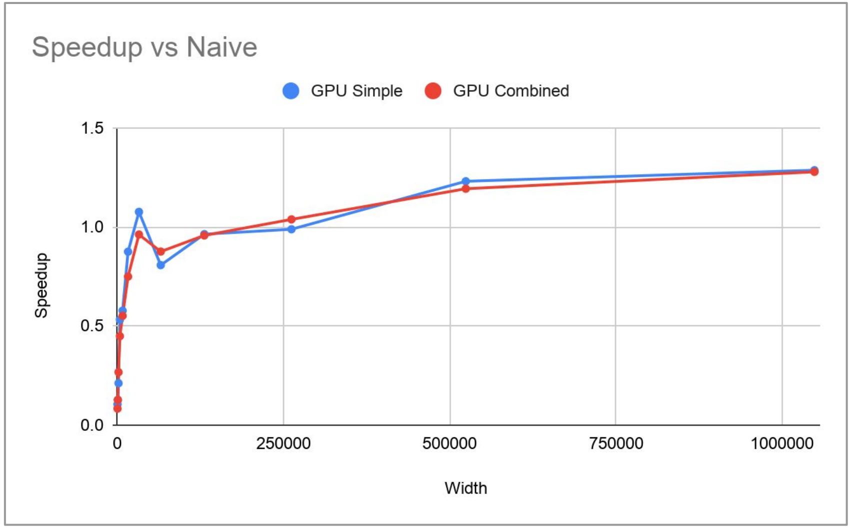
Speedup vs Naive		
Width	GPU Simple	GPU Combined
512	0.1051362265	0.08352877124
1024	0.1276737934	0.1276181398
2048	0.2115546188	0.2671431773
4096	0.5318196596	0.4487701405
8192	0.5775057792	0.551421723
16384	0.8749595489	0.7494225455
32768	1.075764101	0.9615090587
65536	0.8071433045	0.8755240391
131072	0.9633679516	0.9569540645
262144	0.9880425749	1.037878788
524288	1.230122203	1.192645022
1048576	1.28605523	1.27773525

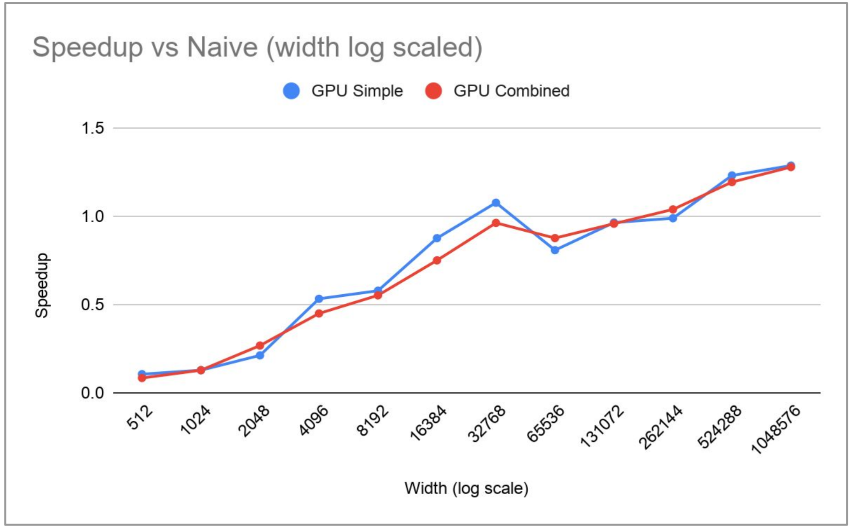
Results (cont.)





Results (cont.)





Conclusions

Current implementation does marginally better at large enough N, more than likely due to heavy communication costs at smaller sizes

Testing-friendly wrappers have an insignificant impact on the average runtime

GPU implementations exhibit large amounts of CPU thrashing, most likely to due repeated re-initializations of starting state with device memory.

Moving Forward:

- Complete Kabsch RMSD calculations require an implementation of the sq. root of a matrix
- Move initialization towards a CUDA kernel to reduce, and perhaps remove, CPU thrashing.

Thoughts? Questions? Ask Away!

References

1. CUDA Programming Guide
    a. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
2. Algorithm
    a. Kabsch, Wolfgang (1976). "A solution for the best rotation to relate two sets of vectors". Acta Crystallographica. A32 (5): 922. Bibcode:1976AcCrA..32..922K. doi:10.1107/S0567739476001873.
3. Wikipedia (for the RMSD image)
    a. https://en.wikipedia.org/wiki/Root-mean-square_deviation_of_atomic_positions